

Department I - C Plus Plus

Modern and Lucid C++
for Professional Programmers

Week 10 – Class Templates

Thomas Corbat / Felix Morgner
Rapperswil, 17.11.2020
HS2020



INSTITUTE FOR
SOFTWARE

- You know when and why to prefer standard algorithms over hand-written loops
- You can name the most important algorithms in the STL
- You can explain certain pitfalls when using STL algorithms
- You can explain the signature of a standard algorithm
- You can write programs that correctly use standard algorithms

- **Recap Week 7**
- **Standard Algorithms**
 - Motivation
 - Algorithm Basics
 - Functors
 - More Algorithm Examples
 - Pitfalls
- **Algorithm Headers**

Recap Week 9



```
template <Template-Parameter-List>  
FunctionDefinition
```

- **Keyword template for declaring a template**
- **Template parameter list: Contains one or more template parameters**
 - A template parameter is a placeholder for a type, which can be used within the template as a type
 - A type template parameter is introduced with the `typename` (or `class`) keyword

- **Example**

Template
Definition

Template
Parameter

```
template <typename T>  
T min(T left, T right) {  
    return left < right ? left : right;  
}
```

- **The compiler...**

- ...resolves the function template
- ...figures out the template argument(s)
- ...instantiates the template for the arguments (creates code with template parameters replaced)
- ...checks the types for correct usage

```
template <typename T>  
T min(T left, T right) {  
    return left < right ? left : right;  
}
```

min(first, second)

TemplateId

Template
Instance

min<int>

```
int min(int left, int right) {  
    return left < right ? left : right;  
}
```

- In specific cases the number of template parameters might not be fix/known upfront

- Thus the template shall take an arbitrary number of parameters

- Example:

```
template<typename First, typename...Types>
void printAll(First const & first, Types const &...rest) {
    std::cout << first;
    if (sizeof...(Types)) {
        std::cout << ", ";
    }
    printAll(rest...);
}
```

- Syntax (ellipses everywhere): ...

- ... in template parameter list for an arbitrary number of template parameters (Template Parameter Pack)
- ... in function parameter list for an arbitrary number of function arguments (Function Parameter Pack)
- ... after sizeof to access the number of elements in template parameter pack
- ... in the variadic template implementation after a pattern (Pack Expansion)

Class Templates



- In addition to functions also class types can have template parameters
- Since C++17, similar to function templates, the compiler might deduce the template arguments

Pre C++17

```
std::vector<int> oldValues{1, 2, 3};
```

C++17

```
std::vector newValues{1, 2, 3};  
std::vector<int> emptyValues{};
```

- Compile-time polymorphism
- Class templates can be specialized

```
template <TemplateParameters>  
class TemplateName { /*...*/};
```

```
template <typename T>  
class Sack { /*...*/};
```

- **A class template provides a type with compile-time parameters**
 - Data members can depend on template parameters
 - Function members are template functions with the class' template parameters
 - Note: Function members can be defined as template member functions with additional template parameters!

```
template <typename T>
class Sack {
    using SackType = std::vector<T>;
    using size_type = typename SackType::size_type;
    SackType theSack{};

public:
    bool empty() const {
        return theSack.empty();
    }
    size_type size() const {
        return theSack.size();
    }
    void putInto(T const & item) {
        theSack.push_back(item);
    }
    T getOut(); //Implementation on the next slide
};
```

Class template with one
typename parameter

Dependent name:
size_type

typename
keyword required

Member function
forward declaration

```
template <typename T>
class Sack {
    using SackType = std::vector<T>;
    using size_type = typename SackType::size_type;
    SackType theSack{};

public:
    //...
    T getOut();
};

template <typename T>
inline T Sack<T>::getOut() {
    if (empty()) {
        throw std::logic_error{"Empty Sack"};
    }
    auto index = static_cast<size_type>(rand() % size());
    T retval{theSack.at(index)};
    theSack.erase(theSack.begin() + index);
    return retval;
}
```

Example for implementing
member functions outside
of a class

Pick random element

```
using Typename = AliasedType;
```

- It is common for template definitions to define type aliases in order to ease their use
 - Less typing and reading
 - Single point to change the aliased type

```
using SackType = std::vector<T>;
```

- Could be a template itself

```
template <typename T>  
using Alias = std::vector<T>;
```

- Old spelling as typedef

```
typedef AliasedType Typename;
```

```
using size_type = typename SackType::size_type;
```

- **Within the template definition you might use names that are directly or indirectly depending on the template parameter**
 - E.g. everything using `SackType::`
- **But you have to tell the compiler if one is a type**
 - In contrast to a variable or function name
- **When the `typename` keyword is required you should extract the type into a type alias**
- **Old spelling in `typedef`**

```
typedef typename SackType::size_type size_type;
```


- Accessing a member of a template parameter

```
template <typename T>
void accessTsMembers() {
    typename T::MemberType m{};
    T::StaticMemberFunction();
    T::StaticMemberVariable;
}
```

```
struct Argument {
    struct MemberType{};
    static void StaticMemberFunction();
    static int StaticMemberVariable;
};
```

- Indirect dependency

```
template<typename T>
class Sack {
    using size_type = typename std::vector<T>::size_type;
    //...
};
```



- There is also a template keyword for a dependent name that is a template. Otherwise, the compiler does not allow to specify template arguments

- Can you tell the concept/requirements for T?

```
template <typename T>
class Sack {
    using SackType = std::vector<T>;
    void putInto(T const & item) {
        theSack.push_back(item);
    }
    T getOut();
};

template <typename T>
inline T Sack<T>::getOut() {
    T retval{theSack.at(index)};
    return retval;
}
```

T needs to be erasable
(implied by std::vector)

T needs to be copyable

T needs to be copyable

- **Members can be defined out of the class template**

- But syntax is a bit ugly!
- They still must be inline, but is implicitly inline as it is a function template

```
template <typename T>
inline T Sack<T>::getOut() {
    //...
}
```

- **Repeat template declaration**

```
template <typename T>
```

- **Keyword inline to avoid ODR violation**

```
inline
```

- **Member Signature**

```
T Sack<T>::getOut()
```

- **Template ID of Sack as name scope**

```
Sack<T>::
```

- **Define class templates completely in header files**
- **Member functions of class templates**
 - Either in class template directly
 - Or as inline function templates in the same header file
- **When using language elements depending directly or indirectly on a template parameter, you must specify typename when it is naming a type**
- **static member variables of a template class can be defined in header without violating ODR, even if included in several compilation units**
 - Since C++17 they can even be declared inside the class template, this requires the inline keyword

```
template <typename T>
struct staticmember {
    inline static int dummy{sizeof(T)};
};
```

TemplateWithStaticMember.h

```
template <typename T>
struct staticmember {
    static int dummy;
};
template<typename T>
int staticmember<T>::dummy{sizeof(T)};
```

SetDummyTo42.cpp

```
#include "TemplateWithStaticMember.h"
int setDummyTo42() {
    using dummytype = staticmember<int>;
    dummytype::dummy = 42;
    return dummytype::dummy;
}
```

Main.cpp

```
#include "TemplateWithStaticMember.h"
#include <iostream>

int main() {
    int setDummyTo42(); //declare setDummyTo42()
    std::cout << staticmember<double>::dummy << '\n';
    std::cout << staticmember<int>::dummy << '\n';
    std::cout << setDummyTo42() << '\n';
    std::cout << staticmember<int>::dummy << '\n';
}
```

● Output:

- 8 (depends on platform)
- 4 (depends on platform)
- 42
- 42

- When class template inherits from another class template name-lookup can be surprising!
- What does the code below print?

```
template <typename T>
struct parent {
    int foo() const {
        return 42;
    }
    static int const bar{43};
};

int foo() {
    return 1;
}

double const bar{3.14};
```

```
template <typename T>
struct gotchas : parent<T> {
    std::string demo() const {
        std::ostringstream result{};
        result << bar << " bar\n";
        result << this->bar << " this->bar\n";
        result << gotchas::bar << " gotchas::bar\n";
        result << foo() << " foo()\n";
        result << this->foo() << " this->foo()\n";
        return result.str();
    }
};
```

3.14 bar
43 this->bar
43 gotchas::bar
1 foo()
42 this->foo()

- **Rule: Always use `this->` or the class name `::` to refer to inherited members in a template class**

```
this->bar
```

```
gotchas::bar
```

- **If the name could be a dependent name the compiler will not look for it when compiling the template definition**
- **Checks might only be made for dependent names at template usage (=template instantiation)**
 - That is sometimes the reason for lengthy error messages from template usages

```
template <typename T>
struct Wrapper {
    T wrapped{};
    void set(T const & value) {
        wrapped.release();
        wrapped = value;
    }
};

Wrapper<int> wrapper{};
```

Correct

As long as you don't call `set()`, the compiler will not recognize, that you cannot call `.release()` on an `int`.

```
template <typename T>
class Sack {
    using SackType = std::vector<T>;
    using size_type = SackType::size_type;
    SackType theSack{};
};
```

Incorrect

`size_type` is a dependent name and therefore has to be declared as `typename` if used as type.

- **Keeping pointers in a sack would require all pointed to variables or objects (pointees) to outlive the sack**
 - This is often hard to achieve
 - And someone must clean up the objects nevertheless
- **Therefore, it might be better to prohibit pointers**

```
Sack<int*> shouldNotCompile;
```

- **Maybe except for character pointers representing string literals, then we can use `std::string` to store them in the Sack**

```
Sack<char const*> shouldKeepStrings;
```

- Like function template overloads, we can provide "template specializations" for class templates
 - These can be **partial** still using a template parameter, but provide (some) arguments
 - Or complete **explicit** specializations, providing all arguments with concrete types
 - One must declare non-specialized template first
 - The most specialized version that fits is used

Partial Specialization

```
template <typename T>  
struct Sack<T *>;
```

Explicit Specialization

```
template <>  
struct Sack<char const *>;
```


- **A class template specialization can have any content, even no content at all**
 - It can be completely unrelated to the original template, there is really no relationship (other than the template name)!
- **A possibility to prohibit instantiating a class is to prohibit the ability to its destruction**
 - Declare its destructor as `= delete`;
 - If an object cannot be destroyed it cannot be created (except for discouraged asymmetrical use of `new`)

```
template <typename T>
struct Sack<T *> {
    ~Sack() = delete;
};
```

```
template <>
class Sack<char const *> {
    using SackType = std::vector<std::string>;
    using size_type = SackType::size_type;
    SackType theSack;
public:
    // no explicit ctor / dtor required
    bool empty() const {return theSack.empty();}

    size_type size() const {return theSack.size();}

    void putInto(char const *item) {
        theSack.push_back(item);
    }

    std::string getOut() {
        if (empty()) {
            throw std::logic_error{"Empty Sack"};
        }
        std::string result = theSack.back();
        theSack.pop_back();
        return result;
    }
};
```

Class template
specialization

No typename keyword,
because it is not dependent

Implementation of
specialization can behave
completely different

- **How can we adapt a standard container by adding invariants or by extending their functionality?**
 - SafeVector -> no undetected out-of-bounds access
 - IndexableSet -> provide operator[]
 - SortedVector -> guarantee sorted order of elements
- **Template class inheriting from template base class**
 - And inherit ctors of standard container
 - Caution: no safe conversion to base class, no polymorphism

```
template<typename T>
struct safeVector : std::vector<T> {
    using container = std::vector<T>;
    using container::container; //or using std::vector<T>::vector;
    using size_type = typename container::size_type;
    using reference = typename container::reference;
    using const_reference = typename container::const_reference;

    reference operator[](size_type index) {
        return this->at(index);
    }

    const_reference operator[](size_type index) const {
        return this->at(index);
    }

    // should also provide front/back with empty() check
};
```

Inherit constructors with using

Add type aliases

Use this-> or container::
prefix for member access

Extending the Sack Template



- Create a Sack<T> using iterators to fill it

```
std::vector values{3, 1, 4, 1, 5, 9, 2, 6};  
Sack<int> aSack{begin(values), end(values)};
```

- Create a Sack<T> of multiple default values

```
Sack<unsigned> aSack(10, 3);
```

- Create a Sack<T> from an initializer list

```
Sack<char> charSack{'a', 'b', 'c'};
```

- Obtain copy of contents in a `std::vector` (for iteration and inspection)

```
Sack<unsigned> aSack{1, 2, 3};  
auto values = static_cast<std::vector<unsigned>>(aSack);
```

- Auto-deducing T for a Sack<T> from an initializer list

```
Sack charSack{'a', 'b', 'c'};
```

```
Sack aSack(begin(values), end(values));
```

- Allow to vary the type of the container to be used

```
Sack<unsigned, std::set> aSack{1, 2, 3};
```

- `std::vector` can be created from a pair of iterators, our **Sack** could do that too

```
void createSackFromIterators() {  
    std::vector values{3, 1, 4, 1, 5, 9, 2, 6};  
    Sack<int> aSack{begin(values), end(values)};  
    ASSERT_EQUAL(values.size(), aSack.size());  
}
```

```
template <typename T>  
class Sack {  
    //...  
public:  
    template <typename Iter>  
    Sack(Iter begin, Iter end) : theSack(begin, end) {}  
    //...  
};
```

Add a constructor template

Use parentheses to avoid calling the
`initializer_list` constructor

- Adding a user-declared constructor removes the implicit default constructor

```
void defaultConstructorStillWorks() {  
    Sack<char> defaultCtor{};  
    ASSERT_EQUAL(0, defaultCtor.size());  
}
```

```
template <typename T>  
class Sack {  
    //...  
public:  
    Sack() = default;  
  
    template <typename Iter>  
    Sack(Iter begin, Iter end) : theSack(begin, end) {}  
    //...  
};
```

Retain default constructor

- This constructor allows the construction with two ints

```
void creationAlsoAllowsTwoInts() {  
    Sack<unsigned> aSack{10, 3};  
    ASSERT_EQUAL(10, aSack.size());  
    ASSERT_EQUAL(3, aSack.getOut());  
}
```

- Our constructor template called with two ints calls the `std::vector(size_type, T const &)` constructor for theSack

- T is unsigned in the case above and the conversion happens implicitly

```
template <typename Iter>  
Sack(Iter begin, Iter end)  
    : theSack(begin, end) {  
}
```



```
Sack(int begin, int end)  
    : theSack(begin, end) {  
}
```

- (Homogeneous) initializer lists `{1, 2, 3}` are passed as the type `std::initializer_list<T>`

```
void createSackFromInitializerList() {  
    Sack<char> charSack{'a', 'b', 'c'};  
    ASSERT_EQUAL(3, charSack.size());  
}
```

- Defining a constructor taking `std::initializer_list<T>` allows us to pre-fill a `Sack<T>`
 - Does not need to be a constructor template
 - Requires `#include <initializer_list>`

```
Sack(std::initializer_list<T> values)  
    : theSack(values) {  
}
```

- This constructor allowing the construction with two ints now behaves differently

```
void creationAlsoAllowsTwoInts() {  
    Sack<unsigned> aSack{10, 3};  
    ASSERT_EQUAL(10, aSack.size());  
    ASSERT_EQUAL(3, aSack.getOut());  
}
```

```
creationAlsoAllowsTwoInts: 10 == aSack.size() expected: 10 but was: 2
```

- List-initialization prefers the `std::initializer_list` constructor
- We can change the initialization to use parentheses to retain the previous behavior

```
void creationAlsoAllowsTwoInts() {  
    Sack<unsigned> aSack(10, 3);  
    ASSERT_EQUAL(10, aSack.size());  
    ASSERT_EQUAL(3, aSack.getOut());  
}
```

```
Sack<unsigned> aSack{1, 2, 3};  
auto values = static_cast<std::vector<unsigned>>(aSack);
```

- Using an explicit conversion operator we can extract a `std::vector` from the `Sack` by copying

```
template <typename Elt>  
explicit operator std::vector<Elt>() const {  
    return std::vector<Elt>(begin(theSack), end(theSack));  
}
```

- Alternatively, a member function template could be implemented

```
template <typename Elt = T>  
auto asVector() const {  
    return std::vector<Elt>(begin(theSack), end(theSack));  
}
```

```
Sack<unsigned> aSack{1, 2, 3};  
auto values = aSack.asVector();  
auto doubleValues = aSack.asVector<double>();
```

- **Considering the Sack template from the previous slides.**

<pre>Sack container{};</pre>	<p>Incorrect</p> <p>The element type cannot be deduced from an empty initializer-list.</p>
<pre>Sack values{0.0, 0.25, 0.5, 0.75, 1.0};</pre>	<p>Correct</p> <p>As we have an <code>std::initializer_list</code> constructor, the compiler can deduce Sack's template argument to be <code>double</code>.</p>
<pre>template <typename T> struct Sack { //... auto asVector() const { return std::vector<T>(begin(theSack), end(theSack)); }; };</pre>	<p>Correct</p> <p><code>asVector</code> does not need to be a member function template. The resulting vector will just always be of type <code>std::vector<T></code>.</p>

Deduction Guides



- Class template arguments can usually be determined by the compiler

```
template <typename T>
struct Box {
    Box(T content)
        : content{content}{}
    T content;
};

int main() {
    Box<int> b0{0}; //Before C++17
    Box      b1{1}; //Since C++17
}
```

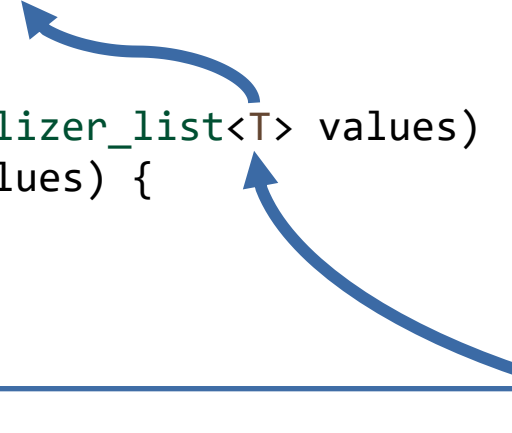
- The behavior is similar to pretending as if there was a factory function for each constructor

```
template <typename T>
Box<T> make_box(T content) {
    return Box<T>{content};
}
```

```
auto gift = make_box(teddy);
```


- In the following example the only template parameter is `T`, which can be deduced from `std::initializer_list<T>`

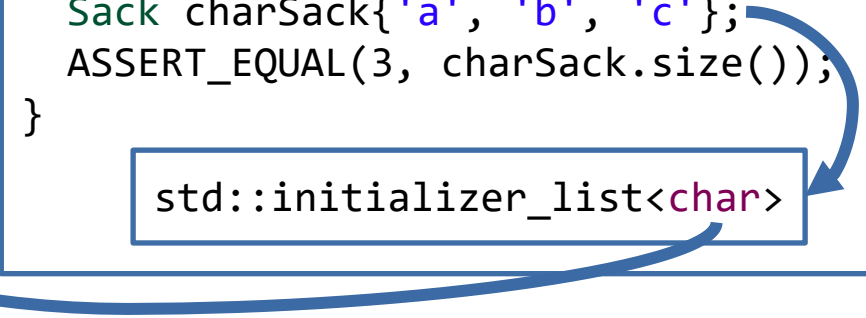
```
template <typename T>
class Sack {
    //...
    Sack(std::initializer_list<T> values)
        : theSack(values) {
    }
    //...
};
```



A blue arrow points from the `T` in the template parameter list to the `T` in the `std::initializer_list<T>` parameter. Another blue arrow points from the `std::initializer_list<T>` parameter to the `std::initializer_list<char>` in the test function.

```
void testImplicitDeductionGuide () {
    Sack charSack{'a', 'b', 'c'};
    ASSERT_EQUAL(3, charSack.size());
}
```

`std::initializer_list<char>`



A blue arrow points from the `charSack` variable to the `std::initializer_list<char>` box. Another blue arrow points from the `std::initializer_list<char>` box to the `std::initializer_list<T>` parameter in the `Sack` class definition.

- There is no direct relation from `Iter` to `T` (constructor parameters to template parameter)

```
template <typename T>
class Sack {
    //...
    template <typename Iter>
    Sack(Iter begin, Iter end) : theSack(begin, end) {}
    //...
};
```

- Why does the following test compile then?

```
void testSurprisingDeduction() {
    std::vector values{3, 1, 4, 1, 5, 9, 2, 6};
    Sack aSack{begin(values), end(values)};
    ASSERT_EQUAL(values.size(), aSack.size());
}
```

Deduces:
`std::vector<int>::iterator`

testSurprisingDeduction: values.size() == aSack.size() expected: 8 but was: 2

- Another attempt, this time with parentheses

```
void testDeductionForIterators() {  
    std::vector values{3, 1, 4, 1, 5, 9, 2, 6};  
    Sack aSack(begin(values), end(values));  
    ASSERT_EQUAL(values.size(), aSack.size());  
}
```

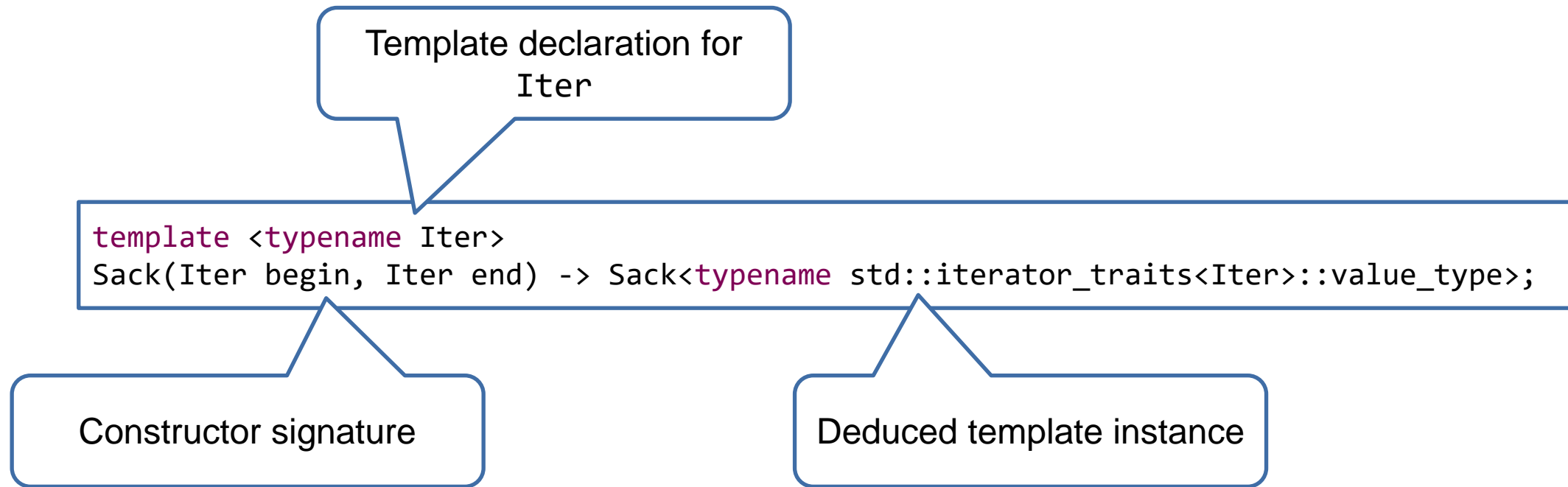
- Results in what we expected

```
error: class template argument deduction failed:  
    Sack aSack(begin(values), end(values));
```

- **User-defined deduction guides can be specified in the same scope as the template**
 - Usually, after the template definition itself

```
TemplateName(ConstructorParameters) -> TemplateID;
```

- **Might be necessary for complex cases, e.g. template constructors if the constructor template parameters do not map directly to the class template parameters**
- **The deduction guide can be (and usually is) a template itself**
- **It looks similar to a free-standing constructor**
- **Unfortunately, C++ does not recognize the deduction guides yet**



- **Test for deducing template argument from iterator works**

```
void testDeductionForIterators() {
    std::vector values{3, 1, 4, 1, 5, 9, 2, 6};
    Sack aSack(begin(values), end(values));
    ASSERT_EQUAL(values.size(), aSack.size());
}
```

- What type will be deduced for the following test?

```
void testDeductionAndMultiDefaultConstruction() {  
    Sack aSack(10, 3u);  
    ASSERT_EQUAL(10, aSack.size());  
    ASSERT_EQUAL(3, aSack.getOut());  
}
```



```
template <typename Iter>  
Sack(Iter begin, Iter end) -> Sack<typename std::iterator_traits<Iter>::value_type>;
```



```
Sack<typename std::iterator_traits<unsigned>::value_type>
```

std::iterator_traits<unsigned>
does not exist

- We need another constructor

```
Sack(size_type n, T const & value)
    : theSack(n, value) {
}
```

- **Is it necessary to write a deduction guide for this constructor?**

No, because the compiler can deduce `T` for `Sack<T>` from the second parameter.

Varying Sack's Container



- A `std::vector` might not be the optimal data structure for a Sack, depending on its usage. E.g., removal from the "middle" requires shifting values in the `std::vector`
- What if we could specify the container type as well as template parameter?

```
Sack<unsigned, std::set> aSack{1, 2, 3};
```

- The implementation of `getOut()` is specific to `std::vector`. We adapt it to more general concept

```
T getOut() {  
    throwIfEmpty();  
    auto index = static_cast<size_type>(rand() % size());  
    auto it = begin(theSack);  
    advance(it, index);  
    T retval{*it};  
    theSack.erase(it);  
    return retval;  
}
```

- A template can take templates as parameters (template template parameters)

```
template<typename T, template<typename> typename Container>  
class Sack;
```

- The template template parameter must specify the number of parameters
- Problem: Standard containers usually take more than just the element type (allocator)

```
Sack<unsigned, std::set> aSack{1, 2, 3}; //Does not work
```

- Pre C++17 the `class` keyword was used

- Unfortunately, C++ does not recognize the `typename` keyword instead of it yet


```
template<typename T, template<typename> class Container>  
class Sack;
```

- We can leave the number of template parameters of the template template parameter unspecified

```
template <typename T, template<typename...> typename Container>  
class Sack;
```

- This allows an arbitrary number of template parameters for the Container parameter

```
Sack<unsigned, std::set> aSack{1, 2, 3}; //Works
```



```
template <typename T, template<typename...> typename Container>  
class Sack;
```

- **The adapted declaration requires to always specify the Container type**

- It breaks all our existing test cases

```
void defaultConstructorStillWorks() {  
    Sack<char> defaultCtor{};  
    ASSERT_EQUAL(0, defaultCtor.size());  
}
```

- **C++ allows default arguments for function and for template parameters**

- This solves our problem

```
template <typename T, template<typename...> typename Container = std::vector>  
class Sack;
```

- **Templates can have non-type parameters**

- Useful for specifying compile-time values
- E.g. the size of a standard array

```
template <typename T, std::size_t n>
auto average(std::array<T, n> const & values) {
    auto sumOfValues = accumulate(begin(values), end(values), 0);
    return sumOfValues / n;
}
```

- **If the type of the non-type template parameter should be flexible auto can be used**

```
template <typename T, auto n>
auto average(std::array<T, n> const & values) {
    auto sumOfValues = accumulate(begin(values), end(values), 0);
    return sumOfValues / n;
}
```

- **Create (partial) specialization if the class template shall behave differently for specific arguments**
- **Specify type aliases to be expressive and have only a single location to adapt them**
- **Access inherited members from other class templates with `this->` or `base::`**
- **Inherit constructors when deriving from a standard container**
- **Deduction guides help the compiler deducing the template arguments**

● Syntax

```
template <TemplateParameters>  
Type TemplateName [= Initializer];
```

- Can be specialized
- Usually constexpr

● Purpose

- Compile-time predicates and properties of types
- Usually applied in template meta programming
- Before C++14 it was necessary to create a class template with a static member variable
 - Now less code is required for the same effect

```
template<typename T>  
constexpr T pi = T(3.1415926535897932385);  
  
template<typename T>  
constexpr bool is_integer = false;  
  
template<>  
constexpr bool is_integer<int> = true;
```

● Before C++14:

```
template<typename T>  
struct numeric_limits {  
    static constexpr bool is_integer = false;  
};  
  
template<>  
struct numeric_limits<int> {  
    static constexpr bool is_integer = true;  
};
```